The Design and Implementation of a Domain-Specific Language for Network Performance Testing

Scott Pakin

Abstract—CONCEPTUAL is a toolset designed specifically to help measure the performance of high-speed interconnection networks such as those used in workstation clusters and parallel computers. It centers around a high-level domain-specific language, which makes it easy for a programmer to express, measure, and report the performance of complex communication patterns. The primary challenge in implementing a compiler for such a language is that the generated code must be extremely efficient so as not to misattribute overhead costs to the messaging library. At the same time, the language itself must not sacrifice expressiveness for compiler efficiency, or there would be little point in using a high-level language for performance testing. This paper describes the CONCEPTUAL language and the CONCEPTUAL compiler's novel code-generation framework. The language provides primitives for a wide variety of idioms needed for performance testing and emphasizes a readable syntax. The core code-generation technique, based on unrolling CONCEPTUAL programs into sequences of communication events, is simple yet enables the efficient implementation of a variety of high-level constructs. The paper further explains how CONCEPTUAL implements time-bounded loops—even those that comprise blocking communication—in the absence of a time-out mechanism as this is a somewhat unique language/implementation feature.

Index Terms—Interprocessor communications, measurement techniques, specialized application languages.

1 Introduction

The performance of parallel applications is determined in large part by the speed of the interconnection network(s) that link together the nodes of a parallel computer or workstation cluster. Understanding parallel-application performance therefore relies on an understanding of the performance of the underlying network. Furthermore, designers of networks and software/firmware messaging layers need to be able to quantify the performance impact of various design modifications. Accurate network performance testing is therefore critical to both the understanding and improvement of overall application performance.

The problem with the way that network performance is currently tested is that testing relies on general-purpose communication benchmarks that provide limited insight into any particular application's performance. These general-purpose benchmarks typically report the performance achievable when combining a small set of communication primitives into a simple communication pattern. In such benchmarks, it is common that both end points are always ready to communicate, there is no intervening computation, and message data is immediately discarded. This behavior is discrepant with that of most complete applications. Although general-purpose benchmarks are useful for demonstrating the communication subsystem's peak performance in a commonly understood format, special-purpose benchmarks targeted to a particular inquiry are an important complement. Unfortunately, special-purpose tests receive little attention in practice and in the literature because they are tedious to write—especially considering that they may be run only a few times before being discarded—and are difficult to explain precisely to others.

CONCEPTUAL—the capitalized letters stand for "Network Correctness and Performance Testing Language"—is a toolset created to facilitate the construction and explication of special-purpose network performance tests. At its core is a domain-specific language that provides primitives for frequently used idioms in communication benchmarks. The language supports collective and point-to-point communication operations (both blocking and nonblocking), precise control over buffer alignment and buffer reuse, support for verifying or simply touching message contents, event timing, statistics calculation, data logging, explicit delays and synthesized computation, command-line parsing, and various other features of relevance to communication benchmarking. The CONCEPTUAL language is designed to read like English-language pseudocode yet remain as precise as any other programming language. Furthermore, CONCEPTUAL abstracts communication away from any particular messaging layer's semantics, ensuring that programs are portable across messaging layers and thereby enabling the comparison of different messaging layers' performance.

This paper's primary technical contributions are:

1. the presentation of a novel compiler code-generation strategy that enables network benchmarks expressed

• *The author is with the Los Alamos National Laboratory, MS B287, Los Alamos, NM 87545. E-mail: pakin@lanl.gov.*

Manuscript received 9 Jan. 2006; revised 20 Sept. 2006; accepted 18 Dec. 2006; published online 9 Jan. 2007.
Recommended for acceptance by R. Eigenmann.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0008-0106.
Digital Object Identifier no. 10.1109/TPDS.2007.1065.

in a high-level domain-specific language to observe no more measurement overhead than those expressed in C, and
2. a demonstration of a language semantics that provides provably deadlock-free and race-free communication, facilitates the expression of complex communication patterns, supports time-bounded iteration, and enables more accurate bit-error reporting than what is possible via a checksum or cyclic redundancy check.

The remainder of this paper is organized as follows: Section 2 describes prior work in the area of domain-specific languages for studying network behavior and explains what makes CONCEPTUAL unique in this space. Section 3 provides further background information about network performance testing and presents some of the challenges inherent in the efficient implementation of a high-level language like CONCEPTUAL. In Section 4, we describe the goals of the CONCEPTUAL language and showcase some of its salient features. The approach currently taken by the CONCEPTUAL compiler is described in Section 5, and the performance of this approach is evaluated in Section 6. Finally, Section 7 draws some conclusions from the information presented in this paper.

2 Related Work

Although domain-specific languages are used for a variety of purposes in the broad category of computer networking, CONCEPTUAL is fairly unique in its use of a domain-specific language to help users measure the performance of high-speed interconnection networks. Anecdotal evidence suggests that a number of small scripting languages for sequencing and timing communication operations have been devised. Such languages tend to be rather ad hoc unpolished creations and are therefore rarely discussed in the literature. One of the few exceptions is MITRE's Local Schedule Executor (LSE) language [1]. LSE is the most closely related system to CONCEPTUAL and can be considered the prior state-of-the-art language for measuring the performance of actual networks. The difference is that LSE's level of abstraction resembles that of an assembly language, whereas CONCEPTUAL provides a rich set of control structures and presents a high-level Single-Program, Multiple-Data (SPMD) [2] view of a parallel system.

At a high level, the Testing and Test Control Notation version 3 (TTCN-3) [3] appears similar in purpose to CONCEPTUAL. Like CONCEPTUAL, TTCN-3 provides language support for performing communication operations and provides support for measuring elapsed time. However, the focus of the two languages is quite different, and this difference is reflected in each language's structure and basic features. In the context of the Open Systems Interconnection (OSI) seven-layer network model [4], TTCN-3's primary purpose is to validate the correctness of the application layer, whereas CONCEPTUAL 's primary purpose, in contrast, is to measure the performance of the transport layer. Hence, CONCEPTUAL provides mechanisms for controlling the reuse, verification, data touching, and memory alignment of message buffers, none of which are supported by TTCN-3—a perfectly valid omission from the perspective of the OSI application layer. However, TTCN-3 provides mechanisms for verifying that a particular request packet results in the reception of a particular reply packet. CONCEPTUAL cannot perform that type of verification because message contents are considered opaque—perfectly correct behavior from the perspective of the OSI transport layer.

Whereas TTCN-3 emphasizes the testing of communication protocols, domain-specific languages such as Prolac [5] have been created to facilitate the implementation of communication protocols, especially those at the OSI transport and network layers. Such languages focus on mechanisms for manipulating message data (extracting headers, computing checksums, reversing byte ordering, and so forth) and other protocol state, but lack, for example, CONCEPTUAL 's mechanisms for calculating performance statistics and logging these to a file.

A third category of domain-specific languages that lie within the broad context of network-related languages includes those that simulate network behavior, generally by using a form of guard conditions and event-triggering messages in the style of Hoare's Communicating Sequential Processes (CSP) [6]. TED [7], Maisie [8], and the NetLanguage language used by REAL [9] are all examples of languages in this category. Although domain-specific network-simulation languages are useful for validating protocol correctness and abstract notions of performance (for example, the number of messages needed to implement a protocol or perform a computation), simulated performance invariably overlooks some of the myriad implementation nuances that can significantly impact network performance, such as the relative alignment of message buffers in the host memory [10]. This is why there exists a need for a language like CONCEPTUAL that measures real-time performance on actual networks.

Outside of the arena of domain-specific languages there are numerous suites of prefabricated performance tests for high-speed interconnection networks. Some of the better known of these include the Intel (formerly Pallas) Message Passing Interface (MPI) Benchmarks [11], SKaMPI [12], NetPIPE [13], and Mpptest [14]. CONCEPTUAL complements prefabricated performance tests by making it easy to explore in detail anomalous or other interesting performance characteristics that are identified by those more general-purpose tests. For example, when one pair of researchers observed suboptimal performance on a particular network benchmark they created a special-purpose benchmark to investigate message-buffer alignment as a potential source of the observed performance loss [10].

3 Background

The goal of a network performance test is to measure the performance of a pattern of communication operations executing in isolation. Network performance tests generally comprise some initialization (allocating message buffers, establishing communication channels between processes, synchronizing processes, and so forth), a call to start the performance timer, calls to the communication operations whose performance is to be measured (usually repeated a

large number of times to amortize timer overhead), a call to stop the performance timer, and some code to compute the performance statistics and log these to a file.

It is critical that noncommunication operations (for example, loop overhead) be minimized to avoid noticeably impacting the measured performance. On today's parallel computers and workstation clusters, process-to-process communication across user-level messaging layers [15], [16] and over a high-speed network can complete in under 3 μs, including the time to traverse all software, firmware, and hardware protocol layers [17]. Hence, a few tens of nanoseconds of overhead can bias measurements by more than a percent.

Most network performance tests are written in C with calls to an MPI library [18], the de facto standard for communication within a parallel application, and measure the performance of some sequence of back-to-back communication operations. Because of the difficulty—or at least, sheer tedium—of expressing in low-level language performance tests that include such practical factors as computational load imbalance or intricate many-to-many communication patterns (possibly involving random peer selection), it is rare that one encounters a nontrivial performance test in practice or in the literature. Nevertheless, more realistic application-centric performance tests are important because they provide insight into application behavior and enable experimentation into alternative communication patterns or varying computation costs that would be more time-consuming to implement and measure in a complete application.

One of CONCEPTUAL's goals is to make it easy to express performance tests that more accurately represent or characterize application behavior. CONCEPTUAL facilitates the expression of arbitrary communication patterns such as wavefronts, N-point stencils, trees, random accesses, and other patterns commonly used by parallel applications. CONCEPTUAL can also simulate computation, thereby enhancing its ability to model an application.

From the perspective of code generation, network performance tests are interesting (and challenging) for the following reasons:

1. Execution speed is critical when the performance timer is ticking, but unimportant otherwise (for example, during program initialization).
2. The important part of a performance test is the calls into the messaging library and other operations that are to be timed. These may not be reordered or optimized away because their order and presence is significant. All other code should be optimized as much as possible so as not to attribute extra overhead to the communication routines.

After highlighting some important aspects of the CONCEPTUAL language's semantics in Section 4, Section 5 describes the CONCEPTUAL code generator and explains how it addresses the preceding issues.

4 Language Design

The driver behind implementing a domain-specific high-level language for network performance testing is to simplify the creation of special-purpose tests that can provide invaluable insight into application behavior. Application-centric network performance tests have to date received little attention from the scientific community partly because the time and effort needed to develop such tests has outweighed the benefits. CONCEPTUAL thereby serves as an enabling technology that finally makes it practical to construct customized tests of network performance.

As a point of terminology, CONCEPTUAL—and for consistency, this article—uses the word "task" to refer more abstractly to a process, thread, node, rank, or whatever else is meaningful to the underlying messaging layer.

4.1 Motivation

The CONCEPTUAL language makes available to the programmer a large number of idioms that facilitate the development of a wide variety of network performance tests. Among these idioms are those for timing the execution of a block of code, calculating statistics, determining neighbor tasks in a variety of (logical) network topologies, transmitting and receiving messages, verifying error-free communication, controlling message-buffer alignment and reuse, parsing the command line, synchronizing tasks, and logging results to files. Although the same functionality could be provided with merely a runtime library, the advantages of using a domain-specific high-level language include:

1. *Portability.* CONCEPTUAL programs are not tied to a particular messaging layer. Hence, the same performance test can be used to compare the performance of disparate messaging layers, for example, to determine the overhead added to a lower-level messaging layer by a more feature-rich higher-level messaging layer.
2. *Readability.* Although the CONCEPTUAL language has a formal syntax and precise semantics, CONCEPTUAL programs read like English-language text, making them a viable alternative to imprecise ad hoc pseudocode for display in a research paper or technical presentation. Furthermore, like a pseudocode description, CONCEPTUAL programs emphasize the pattern of communication, saving the programmer from having to initialize messaging libraries, allocate and fill in data structures, declare variables, keep statistics, or perform other routine activities that are external to the communication pattern being tested. In short, a CONCEPTUAL program is as readable as pseudocode but as precise as the corresponding program coded in a low-level language.
3. *Reproducibility.* By providing within the language the mechanisms needed to record performance data, CONCEPTUAL programs make explicit the set of operations whose performance is to be measured and how the resulting data are to be aggregated and logged to files. The log files that CONCEPTUAL programs produce—described more thoroughly in a prior publication [19]—act like a scientist's laboratory notebook in that they include not only the results of a performance-testing experiment, but also the experimental setup under which that experiment was executed, including any command-line arguments supplied; the mapping from task number to host name; the network interface make and model; the operating system version; the dynamic libraries used;

```
⟨send_stmt⟩      ::= ⟨source_task⟩ [ASYNCHRONOUSLY] SENDS ⟨message_spec⟩ TO [UNSUSPECTING]
                     ⟨target_tasks⟩
                   | ⟨source_task⟩ [ASYNCHRONOUSLY] SENDS ⟨message_spec⟩ TO ⟨target_tasks⟩ WHO
                     RECEIVE IT ⟨recv_message_spec⟩
⟨message_spec⟩   ::= ⟨item_count⟩ [NONUNIQUE | UNIQUE] ⟨item_size⟩ [UNALIGNED | ⟨message_alignment⟩
                     ALIGNED | ⟨message_alignment⟩ MISALIGNED] MESSAGES [WITH VERIFICATION |
                     WITH DATA TOUCHING | WITHOUT VERIFICATION | WITHOUT DATA TOUCHING]
                     [FROM BUFFER ⟨expr⟩ | FROM THE DEFAULT BUFFER]
```

Fig. 1. Sample productions from the CONCEPTUAL grammar.

```
1  ydim is "Mesh height (tasks)" and comes from "--height" or "-h" with default 1.
2  reps is "Number of wavefronts to time" and comes from "--reps" or "-r" with default 1E5.
3  msgsize is "Message size (bytes)" and comes from "--msgsize" or "-m" with default 0.
4
5  For reps repetitions {
6    task 0 resets its counters then
7    all tasks src send a msgsize-byte message to tasks dst such that dst = src+xdim \/ (dst=src+1 /\ dst mod xdim <> 0) then
8    task num_tasks-1 sends a msgsize-byte message to task 0 then
9    task 0 logs the mean of elapsed_usecs/(xdim+ydim-1) as "Per-hop latency (usecs)"
10           and the standard deviation of elapsed_usecs/(xdim+ydim-1) as "Per-hop latency (usecs)"
11 }
```

Fig. 2. CONCEPTUAL implementation of a wavefront algorithm as a network performance test.

the timer type and quality; the CPU count, type, and clock speed; the amount of physical memory; compiler version and options; the username; the timestamp; the environment variables; the complete CONCEPTUAL source program; the termination condition; the total execution time; and the number of interrupts received during the program's execution. In essence, whatever cannot be controlled is at least recorded. The goal is to enable one researcher to completely reproduce another's network performance test given only the log files it generated.

A prior publication [20] argues further why a high-level domain-specific language is a superior alternative to a runtime library coupled with a general-purpose language for the purpose of network performance analysis.

4.2 Grammar

CONCEPTUAL employs a keyword-heavy syntax to help programs read like an English-language description of a communication pattern. Although the language has a formal LALR(1) grammar and therefore requires precision when writing a program, the result can be read and largely understood even by someone unfamiliar with CONCEPTUAL. (The reader is invited to verify that claim by examining the code in Fig. 2.) Although space constraints inhibit the presentation of the complete grammar in this paper, the CONCEPTUAL User's Guide [21] provides the complete Extended Backus-Naur form (EBNF) language specification. The reader is referred to that document to see the language's built-in constructs and evaluate their generality and expressiveness.

At the time of this writing, CONCEPTUAL's EBNF specification includes 56 productions. (For comparison, a typical expression of the American National Standards Institute (ANSI) C grammar includes 64 productions.) Fig. 1 lists, as examples of CONCEPTUAL's grammar, the production for a **send** statement and the production for a message specification. Using those two productions (and a few not shown), one can construct fairly elaborate **send** statements such as "**Tasks** etsk **such that** etsk **is even asynchronously send** 5 **unique page-aligned messages with data touching to tasks** etsk+1 **who receive them synchronously**."[1] Note that the lexer enhances the language's English-like nature by canonicalizing word variants such as **task**/**tasks**, **send**/**sends**, and **it**/**them** and by treating a hyphen as white space before a keyword, as in "**page-aligned**."

4.3 Operational Semantics

One can imagine a CONCEPTUAL program compiling to an n-processor abstract machine. Each processor's state can be represented as a 7-tuple $S_i = (p, B, A, K, C, V, T), 0 \leq i < n$ in which p is the program counter, B is a vector of message queues indexed by the sender for blocking point-to-point messages, A is a vector of message queues indexed by the sender for nonblocking point-to-point messages, K is a vector of message queues indexed by the sender for blocking collective operations (currently, multicasts, reductions, and barrier synchronizations), C is a stack of loop trip counters, V is a stack of queues of loop values, and T is a stack of loop time counters. Although each processor has its own program counter, CONCEPTUAL uses an SPMD semantics. That is, all processors execute the same statements in the same order although possibly skewed in time and possibly effecting different state transitions. In addition to storing each processor's local state, the abstract machine also provides a single wall-clock timer τ, which is shared by all n processors.

Only control structures and communication operations affect the abstract machine's state. All other CONCEPTUAL statements—logging measurements, outputting status information, touching memory, introducing delays, evaluating assertions, and so forth—induce no state changes other than incrementing the program counter.

We define $\mathrm{VPush}(\Sigma_\iota, \eta)$ as the contents of vector Σ after element η is pushed onto stack Σ_ι and $\mathrm{VPop}(\Sigma_\iota, \eta)$ as the contents of vector Σ after the top element η, is popped from

1. This line of code—and, in addition, all future one-line code examples in this paper—is also a complete CONCEPTUAL program.

stack Σ_ι (and discarded). If the stack is empty or the top element is not η, VPop waits until the stack is nonempty, and the top element is η. We further define $\mathrm{Enqueue}(\Gamma, \eta)$ as the contents of queue Γ after element η is pushed onto it, $\mathrm{Push}(\Gamma, \eta)$ as the contents of stack Γ after element η is pushed onto it, $\mathrm{Pop}(\Gamma)$ as the contents of stack or queue Γ after its head is popped (and discarded), and Γ_{top} as the element at the head of stack or queue Γ.

4.3.1 Point-to-Point Communication

Consider the basic CONCEPTUAL **send** statement, which takes a set of source tasks and a set of destination tasks and sends a message m from each source to each destination (as in "**Tasks** abc **such that** abc **is even send a** 1-**megabyte message to tasks** abc+1"). Formally, **send** defines the following sequence of state changes:

1. $\forall s \in sources,\ d \in dests,\ S_d \leftarrow (p, \mathrm{VPop}(B_s, m), A, K, C, V, T)_d$,
2. $\forall s \in sources,\ d \in dests,\ S_d \leftarrow (p, \mathrm{VPush}(B_s, m), A, K, C, V, T)_d$, and
3. $\forall i \in [0, n),\ S_i \leftarrow (p+1, B, A, K, C, V, T)_i$.

Here, m encodes all of the attributes of the message, most notably the sender's processor number and the message size. Step 1 in the preceding list of state changes is omitted if the **unsuspecting** keyword precedes the set of destination tasks (as in "**Task** 0 **sends a** 3-**kilobyte message to unsuspecting tasks** q **such that** $\mathrm{q} > 5$").

Messages sent **asynchronously** (as in, "**All tasks asynchronously send a** 32-**byte-aligned doubleword-sized message to task** numtasks-1") are enqueued on the recipients' nonblocking message queue and not immediately popped: 1) $\forall s \in sources,\ d \in dests,\ S_d \leftarrow (p, B, \mathrm{VPush}(A_s, m), K, C, V, T)_d$, and 2) $\forall i \in [0, n),\ S_i \leftarrow (p+1, B, A, K, C,\ V, T)_i$. Messages sent **asynchronously** are popped by awaiting their completion ("**Task** numtasks-1 **awaits completion**"): 1) $\forall s \in sources,\ d \in dests,\ m \in A_s,\ S_d \leftarrow (p, B, \mathrm{VPop}(A_s, m), K, C, V, T)_d$, and 2) $\forall i \in [0, n), S_i \leftarrow (p+1, B,\ A, K, C, V, T)_i$.

4.3.2 Collective Communication

Collective operations support by CONCEPTUAL—multicasts, reductions, and synchronizations—can be implemented similarly to the point-to-point send operation, but uses the abstract machine's K state instead of the B or A state. All collectives are synchronizing operations and so can be represented by the state changes in the following two-pass ring:

1. $\forall t \in [1, |tasks|)$,
$$S_{tasks_t} \leftarrow (p, B, A, \mathrm{VPop}(K_{tasks_{t-1}}, m), C, V, T)_{tasks_t},$$
2. $\forall t \in [0, |tasks| - 1)$,
$$S_{tasks_t} \leftarrow (p, B, A, \mathrm{VPush}(K_{tasks_{t+1}}, m), C, V, T)_{tasks_t},$$
3. $S_{tasks_0} \leftarrow (p, B, A, \mathrm{VPop}(K_{tasks_{last}}, m), C, V, T)_{tasks_0}$,
4. $\forall t \in [1, |tasks|)$,
$$S_{tasks_t} \leftarrow (p, B, A, \mathrm{VPop}(K_{tasks_{t-1}}, m), C, V, T)_{tasks_t},$$
5. $\forall t \in [0, |tasks| - 1)$,
$$S_{tasks_t} \leftarrow (p, B, A, \mathrm{VPush}(K_{tasks_{t+1}}, m), C, V, T)_{tasks_t},$$
and
6. $\forall i \in [0, n),\ S_i \leftarrow (p+1, B, A, K, C, V, T)_i$.

4.3.3 Control Structures

CONCEPTUAL provides three looping constructs. The simplest performs a fixed number of iterations c, as in "**For** 10 **repetitions** $\langle statement \rangle$." The C state variable counts the number of remaining iterations:

1. $\forall i \in [0, n),\ S_i \leftarrow (p, B, A, K, \mathrm{VPush}(C_i, c), V, T)_i$,
2. recursively perform $\langle statement \rangle$'s state changes,
3. $\forall i \in [0, n),\ S_i \leftarrow (p, B, A, K, C_{\mathrm{top}} - 1, V, T)_i$, and
4. if $C_i = 0$, then $\forall i \in [0, n)$,
$$S_i \leftarrow (p^{(2)}, B, A, K, C, V, T)_i;$$
else, $S_i \leftarrow (p+1, B, A, K, C, V, T)_i$. ($p^{(2)}$ is the address of Step 2.)

CONCEPTUAL's second looping construct iterates over a finite set of values F. An example is "**For each** val **in** {0}, {1, 2, 4, ..., 128K} $\langle statement \rangle$," which assigns to val each value in the set {0} followed by each value in the geometric progression {1, 2, 4, ..., 131,072}. (CONCEPTUAL automatically detects both arithmetic and geometric progressions.) Semantically, this looping construct performs the following operations:

1. $\forall i \in [0, n), v \in F, S_i \leftarrow (p, B, A, K, C, \mathrm{Push}(V, \{\}), T)_i$,
2. $\forall i \in [0, n)$,
$$v \in F, S_i \leftarrow (p, B, A, K, C, \mathrm{Enqueue}(V_{\mathrm{top}}, v), T)_i,$$
3. recursively perform $\langle statement \rangle$'s state changes with the loop variable bound to $V_{\mathrm{top}_{\mathrm{top}}}$,
4. $\forall i \in [0, n),\ S_i \leftarrow (p, B, A, K, C, \mathrm{Pop}(V_{\mathrm{top}}), T)_i$, and
5. if $V_{\mathrm{top}} \neq \oslash$, then $\forall i \in [0, n)$,
$$S_i \leftarrow (p^{(2)}, B, A, K, C, V, T)_i;$$
else, $S_i \leftarrow (p+1, B, A, K, C, \mathrm{Pop}(V), T)_i$.

The third and final looping construct is a timed loop, which runs for a fixed length of time t, as in "**For** 5 **minutes** $\langle statement \rangle$." These loops utilize the abstract machine's global wall-clock timer τ as follows:

1. all processors synchronize (as described above under *Collective Communication*),
2. $\forall i \in [0, n),\ S_i \leftarrow (p, B, A, K, C, V, \mathrm{Push}(T_i, \tau + t))_i$,
3. recursively perform $\langle statement \rangle$'s state changes, and
4. if $\tau < T_{0_{top}}$, then $\forall i \in [0, n)$,
$$S_i \leftarrow (p^{(3)}, B, A, K, C, V, T)_i;$$
else $S_i \leftarrow (p+1, B, A, K, C, V, \mathrm{Pop}(T, T_{i_{top}}))_i$.

4.4 Observations

Some observations one should make from the preceding presentation of CONCEPTUAL's semantics are that

1. all messages sent from s to d are received in order by d,

2. there is no implied ordering of messages sent from two different processors to the same destination,
3. a message must be received explicitly (i.e., there is no "receive any" mechanism), and
4. all loops are executed by all processors and have the same loop bounds on all processors.

The implications of some of these observations are discussed in Sections 4.5 and 4.6.

4.5 Design Decision #1: Language Structure

CONCEPTUAL programs represent finite-state machines. Therefore, CONCEPTUAL is not Turing-complete. It is not possible, for instance, to express an unbounded loop in CONCEPTUAL. There are no variables as such, only let-bindings. Furthermore, a variable cannot be bound to any task-specific value such as a task ID, random number, or length of time. An important ramification of these design decisions is that they enable the compiler to reason about programs in a manner that would not be possible if CONCEPTUAL were Turing-complete. For example, as we shall see in Section 5, the CONCEPTUAL compiler exploits the knowledge that a loop will perform a finite number of iterations and that if there is no explicit loop variable, the loop is guaranteed to execute the same code on every iteration.

Communication, iteration, and measurement are explicit in a CONCEPTUAL program, whereas initialization, buffer allocation, and management of all of the underlying message-passing library's nuances are handled implicitly by the compiler and runtime system. The challenge is in devising a code generator that is so efficient that arbitrary CONCEPTUAL code runs as fast as an equivalent program that is hand-coded in a low-level language and manually structured to minimize the overhead of noncommunication operations. This challenge is the subject of Section 5.

4.6 Design Decision #2: Messaging Semantics

By default, the CONCEPTUAL **send** statement implicitly posts matching receives because this is the common case in many network performance tests. The semantics is such that for a given task, all of the receives introduced by a given **send** statement are posted concurrently before any of the sends.

One important language design decision is that task numbers can be referred to only within simple statements—communication, "computation," logging, and so forth—but not within, for example, let-bindings or conditionals. Because of CONCEPTUAL's SPMD [2] semantics, that decision guarantees that all tasks let-bind the same values to the same variables, follow the same control path and, consequently, agree globally on the sender(s) and receiver(s) for every communication operation. This global agreement has deep implications for programmability in that it provides a simple mechanism for avoiding deadlock.

Proposition 1. *In CONCEPTUAL, no sequence of communication operations in which 1) there are no sends to self and 2) each operation specifies exactly one sender and exactly one receiver can lead to deadlocked communication.*

Proof. Assume that there exists a sequence of point-to-point communication operations

$$\{S_1 \rightarrow R_1, S_2 \rightarrow R_2, \ldots, S_n \rightarrow R_n\},$$

$S_i \neq R_i$ that does lead to communication deadlock. (S_i represents the ith sender, and R_i represents the ith receiver in the sequence of n sends and n receives.) This implies that for some subset of tasks, each task in that subset is blocked at a receive (alternatively, send) waiting for a send (alternatively, receive) from another task; in other words, there exists a cycle of dependencies.

Because CONCEPTUAL ensures that all tasks use the same variable bindings, all tasks therefore attribute the same sender and receiver to a given (static) **send** statement. Because all tasks follow the same control path, all (dynamic) communication operations must be observed and, when applicable, performed by all tasks. As a result of these constraints, all tasks must agree that S_1 is the first sender and R_1 is the first receiver. Therefore, the $S_1 \rightarrow R_1$ communication operation can clearly complete because there are no communication operations ahead of it. Afterward, all tasks must agree that S_2 is the first remaining sender and R_2 is the first remaining receiver, so $S_2 \rightarrow R_2$ can complete. The result is that there exists a total ordering on communication operations. However, a total ordering contradicts the notion that there is a cycle of dependencies (with a cycle implying no ordering). This violates the assumption that deadlock is possible and thereby proves that it is not. □

As an example, expressing ring communication as "**For each** tnum **in** $\{0, \ldots, \text{num\_tasks} - 1\}$ **task** tnum **sends a** 2 **gigabyte message to task** (tnum+1) **mod** num_tasks" guarantees deadlock-free execution as long as the program is run with at least two tasks (so as not to violate the first condition of Proposition 1). However, caution must be exerted when performing multiple communication operations in a single **send** statement. "**All tasks** tnum **send a** 2 **gigabyte message to task** (tnum+1) **mod** num_tasks" *will* deadlock because all of the receives are posted concurrently followed by all of the sends, thereby contradicting the notion of a "first" $S_i \rightarrow R_i$ operation. Similarly, programs can safely perform blocking communication among randomly selected tasks without fear of them deadlocking—an important consequence for programmers who wish to utilize randomness in network performance tests.

A second design decision involving the messaging semantics is that all messages sent from s to d are received by d in the order in which they were sent by s and, furthermore, there are no wildcard receives; to receive a message, a task must specify explicitly the sender and the message size. An important consequence of this construction is that CONCEPTUAL programs are completely deterministic. It is not possible to express a race condition in a CONCEPTUAL program.

Any communication operation that targets a nonexistent task is silently dropped. This design decision reduces the reliance on special cases for edge tasks and simplifies programs, similar to the way that ZPL's "@"-operator [22] simplifies programs by reducing the reliance on special cases for edge elements in array operations. Furthermore, as Section 5 explains, dropped communication operations have no impact on performance. CONCEPTUAL's nonexistent-task semantics implies that a deadlock-free ring can also be expressed with "**All tasks** tnum **send a** 2 **gigabyte message to**

task tnum+1 **then task** num_tasks−1 **sends a** 2 **gigabyte message to task** 0″ even though the first **send** specifies—among other communications—that task num_tasks−1 sends a message to nonexistent task num_tasks.

Finally, messages in CONCEPTUAL are opaque. That is, a program cannot explicitly read or write a message's contents. This design decision gives the runtime system some additional flexibility that is needed to support statements such as "**For** 1E6 **repetitions, task** 0 **sends a** 10 **megabyte message with verification to task** 1 **then task** 1 **logs** bit_errors **as** "Bad bits." " Whereas a checksum or cyclic redundancy check cannot give an accurate count of bit errors, CONCEPTUAL can do so by sending a message with known contents and tallying the number of incorrect bits on the receive side. More precisely, each message sent **with verification** comprises a one-word random-number seed followed by the first $N-1$ random numbers produced using that seed. (CONCEPTUAL uses a 64-bit Mersenne Twister [23] as its random-number generator.) The message recipient initializes its random-number generator using the seed taken from the message, generates $N-1$ random numbers, and performs a bitwise comparison of the two message streams to acquire an accurate bit-error count. Although a corrupted seed will produce false positives, it is virtually impossible for the existence of a bit error to pass undetected when using this scheme.

4.7 A Complete Program

Up to this point, only single-line CONCEPTUAL programs have been presented. In this section, we examine a complete network performance test in some detail. Network performance tests are used for a variety of purposes: 1) prototyping communication algorithms before implementing them in a library, 2) creating application mock-ups to aid the analysis by separating communication and computation costs, and 3) informing performance models, which formulate application scalability as a function of network performance. A typical network performance test, at least in the context of parallel computing, comprises timing measurements of a specified communication pattern. This pattern is generally repeated a number of times to amortize the timer overhead.

Consider a wavefront communication pattern, as is used in deterministic transport applications such as Sweep3D [24]. For the purpose of exposition, we use a simplified version of Sweep3D's communication pattern, as illustrated in Fig. 3 for a 4×5 task grid. In this simplified version, only a single wavefront is propagated through the task grid; the multiple wavefront directions used in the actual Sweep3D code would be implemented analogously.

To express a wavefront communication pattern as a benchmark we specify that task 0 begins the wavefront and the final task in the task grid sends a message back to task 0 to notify it to stop the clock. (For a proper network performance test one cannot assume that clocks are globally synchronized across the entire network.) We further specify that task 0 logs the mean and standard deviation of the per-hop latency across some number of repetitions of this pattern.

The complete CONCEPTUAL code for the wavefront benchmark is presented in Fig. 2. The code in Fig. 2 is typical of a CONCEPTUAL program both in terms of length and complexity. CONCEPTUAL programs rarely need to be longer than a few tens of lines to implement even the most intricate communication patterns used by realistic parallel programs.

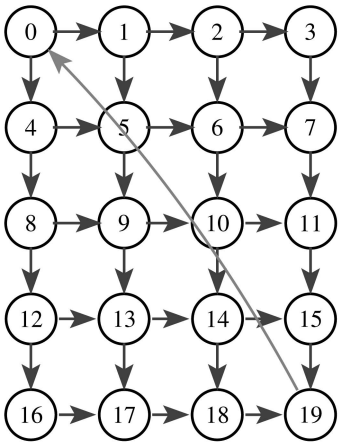

Fig. 3. Wavefront communication pattern.

Each of the first three lines of the program defines a command-line option, including a description to be output if the program is run with `--help`. The program's main loop, lines 5-11, is repeated *reps* times, the default being 1E5 (that is, 10,000), as specified in line 2. Line 6 resets the performance timer and a variety of other counters (bit-error counts and tallies of bytes and messages sent/received).

The entire wavefront communication pattern is implemented in line 7 with line 8 sending a message from the final task back to task 0. In line 7, the "dst = src + xdim" condition makes every task send downward, and the "(dst = src + 1 ∧ dst **mod** xdim <> 0)" condition makes every task except those on the right edge of the task grid send to the right. Note that no special case is needed for the sending-downward condition; CONCEPTUAL automatically suppresses messages sent to nonexistent tasks. In the sending-right condition, task src+1 does exist, so a special case is needed. In practice, line 7 can be simplified by exploiting CONCEPTUAL's built-in mesh_neighbor function; the line is shown as it is to emphasize the generality of CONCEPTUAL's primitive operations.

Lines 9 and 10 calculate the mean and standard deviation of the per-hop latency across all *reps* repetitions and log these values to the disk using the string "Per-hop latency (usecs)" as the primary column header for both columns. A secondary column header—"(mean)" for the first column and "(std. dev.)" for the second column—is generated automatically by CONCEPTUAL. The predeclared elapsed_usecs variable keeps track of the number of microseconds that have elapsed since the beginning of the program or the last **resets its counters** statement.

CONCEPTUAL's English-like syntax makes CONCEPTUAL programs quite readable. Nevertheless, CONCEPTUAL programs may appear excessively verbose to a programmer conversant only with more traditional general-purpose programming languages. However, because CONCEPTUAL's domain-specific semantics supports implicit receives and discarded communication with nonexistent tasks, CONCEPTUAL programs are generally significantly *shorter* than those written in other languages. To help validate this claim, Fig. 4 implements the same functionality as lines 7-8 of the code in Fig. 2, but in C and

```
1  if (dst%xdim!=0)MPI_Recv(m,msgsize,MPI_BYTE,dst-1,1,MPI_COMM_WORLD,&s);if(dst>=xdim)MPI_Recv(m,msgsize
   ,MPI_BYTE,dst-xdim,1,MPI_COMM_WORLD,&s);if((src+1)%xdim!=0)MPI_Send(m,msgsize,MPI_BYTE,src+1,1,MPI_COMM_WORLD
   );if(src<num_tasks-xdim)MPI_Send(m,msgsize,MPI_BYTE,src+xdim,1,MPI_COMM_WORLD);if(dst==0)MPI_Recv(m,msgsize,MPI_BYTE
   ,num_tasks-1,1,MPI_COMM_WORLD,&s);else if(src==num_tasks-1)MPI_Send(m,msgsize,MPI_BYTE,0,1,MPI_COMM_WORLD);
```

Fig. 4. Dense, C+MPI equivalent of Fig. 2, lines 7-8.

using MPI as the communication library—the most common language/library combination for developing tests of communication performance for parallel computers and workstation clusters. The C code is written with no unnecessary spaces so as to minimize its length. Table 1 compares the code in Fig. 4 to that in Fig. 5, which represents lines 7-8 of the code in Fig. 2 with unnecessary spaces removed. According to the table, regardless of whether one counts statements, tokens, or characters, the CONCEPTUAL code is significantly shorter than the corresponding C code. The difference in code length is even greater when comparing versions of the complete wavefront benchmark because the C version has to include header files, declare variables, allocate memory, compute performance statistics, and perform various other minor operations. None of these requires much code by itself, but they add up to a total program length much greater than the equivalent CONCEPTUAL program. This is no accident; CONCEPTUAL is *designed* to facilitate the efficient expression of network performance tests.

5 Implementation

The CONCEPTUAL compiler follows a classic compiler structure: front end, lexer, parser, semantic analyzer, and code generator. CONCEPTUAL supports a variety of code generators from which the user selects at compile time. As of this writing, CONCEPTUAL includes

- a code generator that produces a C code with calls to an MPI [18] library for communication,
- a code generator that produces a C code that communicates using Unix-domain datagram sockets [25],

and a few compiler back ends that are not technically code generators, including

- a CONCEPTUAL interpreter that exploits its global knowledge of the program execution to detect and report deadlocks and other communication bugs in the user's code,
- a back end that uses Dot [26] to visualize the abstract syntax tree corresponding to the input program,
- a back end that specializes any of the C-based back ends, inserting tracing calls (`fprintf()`s or calls to the curses library) into the generated C code,
- a back end that specializes any of the C-based back ends, inserting timing code around every individual communication operation,
- a back end that reports a variety of statistics about a program's execution (for example, network-bisection volume and communication-peer offsets), and
- a back end that uses LaTeX and PSTricks [27] to draw the space-time diagram corresponding to a run of a CONCEPTUAL program.

CONCEPTUAL can theoretically compile to any language/messaging library combination; the preceding list represents only the first batch of back ends that have been developed.

The ability to produce a space-time diagram of a program's execution directly from the CONCEPTUAL source (using the LaTeX + PSTricks back end) has proven to be an immensely useful capability. Subtle bugs in the implementation of a communication pattern become transparent upon examination of a graphical depiction of the communication operations. Also, because CONCEPTUAL is commonly used to implement performance tests that employ nonstandard communication patterns, a visual representation of these patterns is often the most effective way to convey a test's operation.

An implementation decision that is eminently relevant to CONCEPTUAL's acceptance by the networking community is that all of the code generators developed to date output commented human-readable code. It is therefore possible for one to understand not only the CONCEPTUAL source program, but also the generated low-level code, and thereby verify that the benchmark is performing the expected operations.

The rest of this section focuses on the C-based code generators, both of which derive from the same base class—a core C-code generator—and merely specialize it for the appropriate messaging library.

5.1 The Challenge

Consider, for example, the CONCEPTUAL statement "**All tasks** src **send a** 64 **kilobyte message to task** (src − 1)/2," which causes each task in the program to send a message to its parent in a logical binary tree. The hand-coded C equivalent would likely begin with each task determining whether it is a leaf, a 1-child internal node, a 2-child internal node, a 1-child root, or a 2-child root and branch to one of the five main timing loops, as appropriate. By hoisting the child and parent tallies out of the timing loop, each loop body can hardwire the number of sends and receives to post, thereby minimizing the execution-time overhead. Although it is easy for a human to enumerate all of the possible alternatives and specialize the communication

TABLE 1
Comparison of Fig. 4 and Fig. 5

Metric	C	CONCEPTUAL
Statements	6	2
Tokens	158	45
Characters	432	170

```
1 all tasks src send a msgsize−byte message to tasks dst such that dst=src+xdim\/(dst=src+1/\dst mod xdim<>0) then task num_tasks
  −1sends a msgsize−byte message to task 0
```

Fig. 5. Dense version of Fig. 2, lines 7-8.

code for each alternative, it is far more challenging for a compiler to do likewise.

As programs get more complicated than the one-liner shown above, it becomes difficult even for a human to structure a program in such a way as to hoist the overhead out of the timing loop. Consider the CONCEPTUAL statement "**For** 1,000 **repetitions**, **let** aaa **be a random task and** bbb **be a random task other than** aaa **while task** aaa **sends a** 256 **byte message to task** bbb." Random communication patterns that use blocking communication are problematic for performance tests for two reasons. First, the random-number generator must be globally coordinated across all tasks in the program to prevent a mismatched send or receive from hanging the program. (Messaging layers such as MPI [18] lack a time-out mechanism.) Second, care must be taken to avoid deadlock situations because the random selection of communication end points can produce a cycle of blocked tasks if this situation is not explicitly prevented. Third, random-number generation is a comparatively slow operation; a large component of the reported communication time may in fact be attributable to random-number generation time. There is virtually no history of performance tests that employ blocking communication among randomly selected end points. The parallel version of the Apex-Map benchmark [28], for example, uses exclusively nonblocking random communication and employs a cleanup phase at the end to force the completion of all mismatched sends and receives.

5.2 Initial Attempt

An early implementation of the CONCEPTUAL compiler took a straightforward approach to code generation: CONCEPTUAL arithmetic expressions produced C arithmetic expressions, CONCEPTUAL communication operations produced calls to a messaging library's communication operations, CONCEPTUAL loops produced C loops, and so forth. For simple programs, this simple approach worked well and generated a code that was fairly similar in both style and performance to the hand-coded C. Unfortunately, the approach failed to scale with the complexity of the input. When executing a CONCEPTUAL statement such as "**For each** ofs **in** {1, 2} **all tasks** sndr **send a** 64-**byte page-aligned message to task** 2*(sndr **mod** 5) + ofs" with a total of eight tasks, for example, each task will send zero, one, or two messages and receive 0, 1, or 2 messages. (As mentioned in Section 4.6, no messages are sent to or received from nonexistent tasks.) Because there is an arbitrary relation between senders and receivers and because the number of tasks is defined at runtime, the compiler does not know how many sends or receives to post. These numbers can be calculated at runtime, but doing so essentially requires that each task evaluate "2*(sndr **mod** 5) + ofs" for *every* value of sndr—a costly amount of overhead to incur while communication time is being measured.

5.3 The Improved Approach

The key observation to make is that the generated code needs to be efficient *only while performance is being measured*. Hence, any overhead operations (that is, those not related to communication or otherwise relevant to the timing measurements) can be hoisted above the timed blocks of the code to program initialization time. This is in contrast to the typical manner in which compiler performance is evaluated in which the entire program's execution time is considered. In the rest of this paper, we refer to the period before the performance timer is started as *initialization time* and the period afterward as *execution time*.

Given that the performance of program initialization is immaterial (within reason) and that the CONCEPTUAL language is not Turing-complete, it is possible to "preexecute" a CONCEPTUAL program at initialization time. In essence, the compiler performs a partial evaluation of the input program with respect to the number of tasks and command-line arguments and completes the compilation at initialization time. During the preexecution phase, the generated code determines the precise set of operations that will need to be performed once the timer starts ticking and creates a list of events (send, receive, synchronize, multicast, "compute," log to file, and so forth) for the calling task to execute. For example, if the CONCEPTUAL statement "**Tasks** nz **such that** nz **is odd send** 5 3-**doubleword-sized messages to task** 0 **who receives them with data touching**" is run with 10 tasks, the event lists of tasks 1, 3, 5, and 9 will each contain five send events, each specifying the transmission of a 3-doubleword (24-byte) message to task 0; task 0's event list will contain 20 receive events, each specifying the receipt of a 3-doubleword message whose data must be touched (read and written) after receipt to stress the memory hierarchy.

After a task constructs its list of events, it starts the timer and begins the program's main loop, which simply iterates over each event in turn and executes whatever code is appropriate to the target messaging layer. Fig. 6 presents a typical instantiation of the event loop as generated by the C+MPI back end. (The C+sockets back end, for example, calls `send()` instead of `MPI_Send()`.) Only events actually utilized by the CONCEPTUAL program appear as cases in the **switch** statement.

To improve memory utilization, CONCEPTUAL's "**for** ⟨*expr*⟩ **repetitions** ⟨*statement*⟩" construct produces a repeat event followed by a single instantiation of the loop body rather than producing ⟨*expr*⟩ complete instantiations of the loop body. (Note the recursive invocation of `conc_process_events()` in Fig. 6.) This optimization is possible because there is no loop variable in "**for** ⟨*expr*⟩ **repetitions** ⟨*statement*⟩"; hence, the loop body is invariant across iterations. In contrast, CONCEPTUAL's "**for each** ⟨*var*⟩ **in** ⟨*set*⟩ ⟨*statement*⟩" construct can result in a different set of events for each iteration and is therefore ineligible for this memory optimization. In practice, however, the size of ⟨*set*⟩ tends to be comparatively small, so optimizing memory usage is less important in that context.

The following are some of the benefits of expanding a CONCEPTUAL program into an event list at initialization time:

```
1  static void conc_process_events (CONC_EVENT *eventlist, ncptl_int firstev, ncptl_int lastev, ncptl_int numreps)
2  {
3    CONC_EVENT *thisev; /* Cache of the current event */
4    CONC_EVENT *thisev_first = &eventlist[firstev]; /* Cache of the first event */
5    ncptl_int i, j; /* Iterate over events and repetitions. */
6    MPI_Status status; /* Not needed but required by MPI_Recv() */
7
8    /* Process from event firstev to event lastev (both inclusive). */
9    for (j = numreps; j > 0; j--)
10     for (i = firstev, thisev = thisev_first; i <= lastev; i++, thisev++) {
11       switch (thisev->type) {
12         case EV_SEND:
13           MPI_Send (thisev->s.send.buffer, (int) thisev->s.send.size, MPI_BYTE, (int) thisev->s.send.dest, 0, MPI_COMM_WORLD);
14           break;
15         case EV_RECV:
16           MPI_Recv (thisev->s.recv.buffer, (int) thisev->s.recv.size, MPI_BYTE, (int) thisev->s.recv.source, 0, MPI_COMM_WORLD,
                 &status);
17           break;
18         case EV_SYNC:
19           MPI_Barrier (thisev->s.sync.communicator);
20           break;
21         case EV_REPEAT:
22           conc_process_events (eventlist, i+1, thisev->s.rep.end_event, thisev->s.rep.numreps);
23           i = thisev->s.rep.end_event;
24           thisev = &eventlist[i];
25           break;
26                                                                    ⋮
27       }
28     }
29 }
```

Fig. 6. Sample event loop in the CONCEPTUAL-generated C code, specialized for MPI.

- There is a consistent mechanism for hoisting events from the timed part of the performance test to the untimed initialization part.
- All expressions, no matter how complex, take zero time to evaluate from the perspective of the timed part of the code. A corollary is that checking for a peer task's existence (for example, in the context of a **send** statement) takes zero time from the perspective of the timed part of the code.
- Any operation that at execution time will not apply to a particular task is automatically elided from that task's event list. Hence, there is no execution-time cost to interleaving statements applicable to one set of tasks with statements applicable to a different set of tasks.
- CONCEPTUAL's functionality can easily be enhanced by introducing new events or adding new fields to existing events—even on a per-back-end basis. For example, the execution-tracing back end appends a pair of line-number fields to every event structure and injects a code into the main loop to output the name of each event, a few event parameters, and the lines of the source code to which that event corresponds.

As a specific example of how an event-based compiler can reduce the measurement overhead, recall that CONCEPTUAL enables source and destination tasks to be selected using a random-number generator, as showcased in Section 5.1. Although random-number generation is a comparatively slow operation, the cost is hoisted to initialization time and therefore imposes no more penalty at performance-testing time than would the use of constants.

A related approach could be to create an event list at compile time instead of initialization time. Doing so could reduce the overhead by replacing Fig. 6's **switch** statement and **for** loops with straight-line calls to the underlying messaging layer. However, this approach would either restrict each CONCEPTUAL program to a number of tasks selected at compile time or require the generation of numerous **if** statements to check at runtime for communication with nonexistent tasks. Consider "**All tasks** ff **send a message to task** ff–5"; how many receives should task 0 post?

5.4 Timed Loops

Performance tests that run for a bounded length of time rather than a bounded number of iterations are more common in the context of Internet networking than in the context of parallel-computer or cluster networking. Nevertheless, CONCEPTUAL does provide support for such tests in the form of a timed-loop construct, as in "**for** $\langle time \rangle$ $\langle statement \rangle$." CONCEPTUAL executes the loop body for the corresponding length of real time. This is a best effort operation; a loop iteration is not preempted before completing.

The challenge of using real-time loop bounds in the context of network performance tests is that it can lead to send-receive mismatches that are problematic when employing blocking communication. Consider, for example, the case in which task 0 is sending messages to task 1 for a given length of time. If task 0's timer expires after posting N sends but task 1's timer expires after posting $N + 1$ receives, then task 1 will block indefinitely waiting for a message that will never arrive. This scenario can occur even if all tasks are perfectly time synchronized because it may take different lengths of time to perform a send versus a receive operation. Existing solutions from other contexts include using out-of-band data to signal performance-test termination or using a message-time-out mechanism to prematurely terminate a blocking communication operation. The

former approach is used, for example, by the Netperf performance-test suite [29], and the latter is used, for example, by the Erlang parallel-programming language [30]. Unfortunately, neither approach can be integrated comfortably into CONCEPTUAL because some messaging layers (most notably, MPI [18]) lack both out-of-band data transmissions and message time outs and CONCEPTUAL's intention is to be portable across a wide variety of messaging layers.

Instead, the approach taken by the CONCEPTUAL implementation is as follows: First, all tasks stop their performance counters (that is, message counters, byte counters, the bit-error counter, and the performance timer) and enter a mode in which nonidempotent operations such as writing to the log file are suppressed. Then, each task measures the time needed to perform a fixed number of trial loop iterations. Because the timed-loop construct does not employ a loop variable, all iterations are necessarily identical and can therefore be assumed to take approximately equal lengths of time. Next, all tasks obtain global agreement on the number of iterations that can execute in time $\langle time \rangle$ by taking the maximum iteration estimate across all tasks. An overshoot factor is added to this number so as to better tolerate performance variation. Once all tasks have agreed upon the number of iterations to perform, each task reenables the execution of nonidempotent operations, restarts its performance counters, and performs the agreed-upon number of iterations. After each loop iteration, each task checks if time $\langle time \rangle$ has elapsed. If so, the task stops its performance counters and once again suppresses nonidempotent operations, but continues running until all iterations have completed. Once all tasks have exited the loop, they synchronize and return to their normal execution mode.

The preceding approach achieves the goal of iterating for a bounded length of time without sacrificing portability across messaging layers or support for blocking communication operations. Each task's performance timer "sees" the correct length of time although more iterations may actually have been performed. Like the core CONCEPTUAL implementation structure described in Section 5.3, the implementation of timed loops exploits the underlying assumption that performance is important only during the timed part of a program's execution. Before the performance timer starts and while the timer is paused, extra operations can be executed to reduce the overhead or enhance functionality without incorrectly attributing their cost to communication.

6 Evaluation

In this section, we evaluate the approach taken by the CONCEPTUAL compiler, namely, the compilation of network performance tests into a code that evaluates expressions and loops at program initialization time and merely executes each event in a generated event list during execution time. The metric on which we choose to focus is the overhead reported in the timing measurements. That is, for a given performance test, we wish to determine how much worse the CONCEPTUAL-reported performance is than that reported by a hand-coded C program.

To make the comparison as stringent as possible, we utilize a communication latency test as our benchmark. A latency test is, in spirit, the networking analog of a compiler benchmark that computes the nth Fibonacci number: Both programs are so simple that they offer a high-level language compiler no opportunity to improve performance over that realized by a low-level language but myriad opportunities to degrade performance. The goal of a latency test is to measure the time to send a message of a given size from one task to another and back. The test reports half of that round-trip time as an estimate of the one-way time. (Clocks are rarely globally synchronized—especially not to submicrosecond granularity—so one-way latency cannot generally be measured directly.) Because there is no overlap of communication operations and because all communication is blocking, all overhead introduced by the compiler is exposed in the timing measurements. To further emphasize any inefficiencies in the CONCEPTUAL-generated code, we measure the latency only of small messages—those consisting of merely 0-32 bytes of payload. Doing so prevents code overheads from appearing insignificant relative to large-message transmission times.

There are numerous ways to construct a latency test; subtle variations in implementation can result in dramatically different performance being reported [19]. To avoid biasing the selection of a particular latency test toward one that may lend itself well to an efficient CONCEPTUAL implementation, we started with an existing latency program that has been used to measure performance on a variety of networks, is freely available on the Web, has been discussed in the literature [31], and originates from an established research group bearing no connection to that of the author.

The original code, written in C with calls to an MPI library [18], is available at http://nowlab.cse.ohio-state.edu/projects/mpi-iba/performance/osu_latency.c. The CONCEPTUAL equivalent is presented in its entirety in Fig. 7. Although the two versions of the latency test perform identical communication operations and use the same number and alignment of message buffers, the CONCEPTUAL version requires only a third as much code as the C version yet is strictly more featureful. The CONCEPTUAL version accepts command-line options where the original uses hardwired constants; it verifies that there are at least two tasks available and outputs a helpful error message if not; and it logs not only the performance measurements, but also enough information about the execution environment to facilitate the reproduction of the experiment and results by a third party [19]. The question, therefore, is how faithful the CONCEPTUAL-generated performance measurements are to those reported by the original C code.

We compared the performance reported by the hand-coded C+MPI version of the latency test to that reported by the CONCEPTUAL version compiled using the C+MPI code generator. Both the hand-coded and generated C programs were compiled with version 7.1 of Intel's `ecc` compiler with optimization level `-O3`. All tests were run across a pair of dual 1.3-GHz Itanium II nodes interconnected with a Quadrics QsNet network [32]. For each of the two latency-test implementations, latency was measured 11 times per message size. Fig. 8 plots all 11 trials for each data point with a line running through the arithmetic means and each mean stated numerically. That figure clearly demonstrates that there is no qualitative difference between the performances reported by the two versions of the latency test. More formally, from a statistics perspective, no conclusions can be drawn from Fig. 8's data with 99 percent confidence. With 95 percent confidence, however, one can conclude that the CONCEPTUAL version of the test does observe a slight amount of additional overhead relative to the C version. This overhead—reported here as the ratio of the CONCEPTUAL

```
1  # ------------------------------------------------------------
2  # coNCePTuaL version of the OSU MPI latency test, version 2.0
3  # ------------------------------------------------------------
4
5  # Hardwired constants in the original are replaced with command-line options here.
6  message_alignment is "Message buffer alignment (bytes)" and comes from "--align" or "-a" with default 64.
7  max_msg_size is "Maximum message size (bytes)" and comes from "--max-size" or "-m" with default 4M.
8  skip is "Number of small-message warmup iterations" and comes from "--skip" or "-k" with default 1000.
9  loop is "Number of small-message iterations" and comes from "--loop" or "-l" with default 10000.
10 skip_large is "Number of large-message warmup iterations" and comes from "--skip-large" or "-j" with default 10.
11 loop_large is "Number of large-message iterations" and comes from "--loop-large" or "-i" with default 100.
12 large_message_size is "Large message size (bytes)" and comes from "--large-size" or "-s" with default 8K.
13
14 # We add this for the user's convenience. The original crashes if given too few tasks.
15 Assert that "The latency test requires at least two nodes" with num_tasks>=2.
16
17 # coNCePTuaL normally recycles message buffers for non-concurrent operations. Here, we let-bind some buffer numbers which
18 # we use below to force coNCePTuaL to use a different buffer for sending and receiving, as in the original C code.
19 Let s_buf be 0 and r_buf be 1 while {
20   for each size in {0}, {1, 2, 4, ..., max_msg_size} {
21     # Warm up the buffers by sending a message "with data touching".
22     task 0 sends a size-byte message_alignment-byte-aligned message with data touching from buffer s_buf to task 1 who receives it
         into buffer r_buf then
23     task 1 sends a size-byte message_alignment-byte-aligned message with data touching from buffer s_buf to task 0 who receives it
         into buffer r_buf then
24
25     # As in the original, the number of iterations depends on the message size.
26     let loop be loop_large if size > large_message_size otherwise loop
27     and skip be skip_large if size > large_message_size otherwise skip while {
28       # Here's the important part of the code, in which we measure the time needed to perform a large number of ping-pongs.
29       all tasks synchronize then
30       task 0 resets its counters then
31       for loop repetitions plus skip warmup repetitions {
32         task 0 sends a size-byte message_alignment-byte-aligned message from buffer s_buf to task 1 who receives it into buffer
             r_buf then
33         task 1 sends a size-byte message_alignment-byte-aligned message from buffer s_buf to task 0 who receives it into buffer
             r_buf
34       } then
35       task 0 logs size as "Size" and elapsed_usecs/total_msgs as "Latency (us)"
36     }
37   }
38 }
```

Fig. 7. CONCEPTUAL version of `osu_latency.c`.

mean latency to the C version's mean latency— corresponds to a slowdown of 0.5-0.8 percent. In absolute terms, the performance difference ranges from 25 to 40 ns (33 to 52 CPU cycles at 1.3 GHz).

Analyzing the generated assembly code of the two latency-test versions reveals that the extra overhead observed by the CONCEPTUAL version is caused by loop start-up overhead in a small-trip event loop (line 10 of Fig. 6), a **switch** statement (line 11), and the extraction of function arguments from a C **struct** (in lines 13, 16, and 19). Commenting out all of the communication and logging code from the event-processing loop while leaving the overhead code—the loops and **switch** statement—results in an average measurement of 37 ns (48 CPU cycles) of overhead per event. As stated earlier in this section, a small-message latency test offers no opportunity to hide the overhead, implying that these slowdowns represent an upper bound in the performance loss induced by the CONCEPTUAL-generated overhead.

Note that none of these overheads is inherent; a later version of the CONCEPTUAL compiler may use loop unrolling to amortize loop start-up costs and may also coalesce events to amortize `switch()` costs. The encouraging point is that even in its initial naïve implementation of event lists, the CONCEPTUAL compiler produces a C code that is extremely competitive in terms of performance with the handwritten C code.

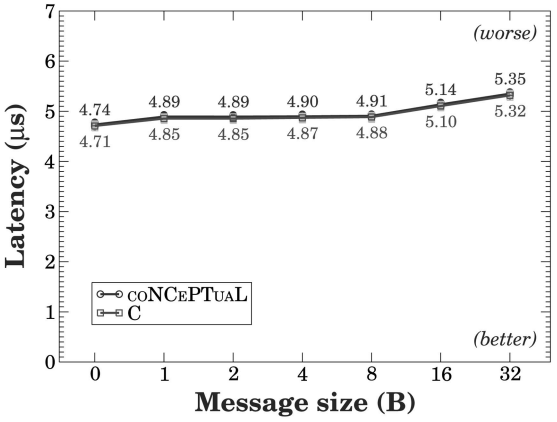

Fig. 8. Latency as a function of message size.

7 CONCLUSIONS

The CONCEPTUAL toolset defines a high-level domain-specific language designed explicitly to simplify the

construction, presentation, and execution of network performance tests. CONCEPTUAL places particular emphasis on tests of high-speed parallel-computer and workstation-cluster interconnects and user-level messaging layers [16]. The challenge in this context is that performance tests are far more sensitive to overhead than they are in the context of distributed, client-server, Grid, or other loosely coupled systems. To minimize the overhead from noncommunication operations, network performance tests are traditionally written in hand-coded C with calls into a particular messaging library, thereby sacrificing program readability (because of explicit program initialization, buffer allocation, and other distractions that must be present when writing a code in a general-purpose language), ease of expressing complex communication patterns, and portability to other languages/messaging libraries. CONCEPTUAL's goal is to address all of these issues while still reporting the same performance as would a performance test coded in a low-level general-purpose language.

From the data presented in Section 6, we can conclude that CONCEPTUAL has very nearly achieved its goal. We compared C and CONCEPTUAL performance on a small-message latency test, which offers no opportunity to hide the noncommunication overhead behind communication operations and, therefore, represents a worst-case scenario for a high-level language compiler. The results show that there is no qualitative difference between the hand-coded and CONCEPTUAL-generated versions of this performance test. Quantitatively, the CONCEPTUAL-generated C program exhibits only a fraction of a percent more overhead than its hand-coded equivalent. Although this overhead is not to be ignored, we do believe that the benefits offered by CONCEPTUAL serve as a viable counterweight to a miniscule performance loss in a worst-case scenario. We expect that future versions of the CONCEPTUAL compiler will close the performance gap.

The key insight needed to develop an efficient compiler for CONCEPTUAL is that execution speed is critical only while measuring network performance. Consequently, inherently slow routines such as evaluating complicated expressions, generating random numbers, determining the relation of senders to receivers, and finding patterns in loop arguments (for example, that " {1, 2, 4, . . . , max_msg_size}" in line 20 of Fig. 7 represents a geometric sequence) can all be hoisted to initialization time at which point execution speed is largely irrelevant. While communication performance is being measured, only communication operations and loops are performed, as is generally the case in a hand-coded network performance test. The specific application of the aforementioned insight as presented in this paper is to unroll a program into a list of events at initialization time and merely walk that list at execution time. Event lists provide a uniform operation-hoisting mechanism that simplifies compiler construction and enables compiler back ends (such as CONCEPTUAL's execution-tracing back end) to specialize other back ends in a consistent manner.

The conclusion that one should draw from this work is that a careful integration of a language semantics and code-generation strategy can make even a performance-critical task such as network performance testing feasible to express in a high-level language. CONCEPTUAL's code generator is able to exploit the distinction between time-sensitive and time-insensitive operations to reduce the arbitrary benchmark programs' measurement overhead to a level comparable to that which a human can achieve by custom-optimizing each individual benchmark program. CONCEPTUAL represents a clearer, more concise, and more expressive approach to developing network performance tests than is otherwise possible and, as this paper demonstrates, can do so with no noticeable degradation in performance.

The CONCEPTUAL source code and documentation are freely available at http://conceptual.sourceforge.net/.

REFERENCES

[1] L. Monk, R. Games, J. Ramsdell, A. Kanevsky, C. Brown, and P. Lee, "Real-Time Communications Scheduling: Final Report," Technical Report MTR 97B0000069, The MITRE Corp., Center for Integrated Intelligence Systems, Bedford, Mass., http://www.mitre.org/tech/hpc/pdf/rtcs_final.pdf, May 1997.

[2] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister, "A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN," *Parallel Computing*, vol. 7, no. 1, pp. 11-24, Apr. 1988.

[3] *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, European Technological Standards Inst., Sophia Antipolis Cedex, http://www.ttcn-3.org/Specifications.htm, 2005.

[4] H. Zimmermann, "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Comm.*, vol. 28, no. 4, pp. 425-432, http://www.comsoc.org/livepubs/50_journals/pdf/RightsManagement_eid=136833.pdf, Apr. 1980.

[5] E. Kohler, M.F. Kaashoek, and D.R. Montgomery, "A Readable TCP in the Prolac Protocol Language," *Proc. ACM SIGCOMM '99*, pp. 3-13, http://www.pdos.lcs.mit.edu/prolac/sigcomm99.pdf, 1999.

[6] C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[7] K. Perumalla, A. Ogielski, and R. Fujimoto, "TeD—A Language for Modeling Telecommunication Networks," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 25, no. 4, pp. 4-11, http://www.cc.gatech.edu/computing/pads/PAPERS/ted-sigmetrics.ps, Mar. 1998.

[8] R.L. Bagrodia and W.-T. Liao, "Maisie: A Language for the Design of Efficient Discrete-Event Simulations," *IEEE Trans. Software Eng.*, vol. 20, no. 4, pp. 225-238, http://pcl.cs.ucla.edu/papers/files/maisie-tse.ps.gz, Apr. 1994.

[9] S. Keshav, "REAL: A Network Simulator," Technical Report CSD-88-472, Computer Science Division, Univ. of California, Berkeley, http://sunsite.berkeley.edu:80/Dienst/Repository/2.0/Body/ncstrl.ucb/CS D-88-472/pdf, Dec. 1988.

[10] L. Arber and S. Pakin, "The Impact of Message-Buffer Alignment on Communication Performance," *Parallel Processing Letters*, vol. 15, no. 1, pp. 49-65, http://www.c3.lanl.gov/pal/publications/papers/Arber2005:alignment.pdf, Mar. 2005.

[11] *Intel MPI Benchmarks: Users Guide and Methodology Description*, Intel GmbH, http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm, Nov. 2004.

[12] R. Reussner, P. Sanders, L. Prechelt, and M. Müller, "SKaMPI: A Detailed, Accurate MPI Benchmark," *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proc. Fifth European PVM/MPI Users' Group Meeting (EuroPVM/MPI '98)*, pp. 52-59, http://www.mpi-sb.mpg.de/sanders/papers/europvm-mpi98.ps.gz, Sept. 1998.

[13] Q.O. Snell, A.R. Mikler, and J.L. Gustafson, "NetPIPE: A Network Protocol Independent Performance Evaluator," *Proc. IASTED/ISMM Int'l Conf. Intelligent Information Management Systems*, http://www.scl.ameslab.gov/netpipe/paper/netpipe.ps, June 1996.

[14] W. Gropp and E. Lusk, "Reproducible Measurements of MPI Performance Characteristics," *Recent Advances in Parallel Virtual Machine and Message Passing Interface: Proc. Sixth European PVM/MPI Users' Group Meeting (EuroPVM/MPI '99)*, pp. 11-18, http://www.mcs.anl.gov/gropp/bib/papers/1999/pvmmpi99/mpptest.pdf, Sept. 1999.

[15] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin, "User-Space Communication: A Quantitative Study," *Proc. ACM/IEEE Conf. Supercomputing (SC '98),* http://www.cs.princeton.edu/dubnicki/papers/araki98userspace.pdf, 1998.

[16] R.A.F. Bhoedjang, T. Rühl, and H.E. Bal, "User-Level Network Interface Protocols," *Computer,* vol. 31, no. 11, pp. 53-60, http://www.cs.cornell.edu/raoul/papers/computer98.pdf, Nov. 1998.

[17] J. Beecroft, D. Addison, F. Petrini, and M. McLaren, "Quadrics QsNet II: A Network for Supercomputing Applications," *Proc. Hot Chips 15 Conf.,* http://hpc.pnl.gov/people/fabrizio/papers/hot03.pdf, Aug. 2003.

[18] *MPI: A Message-Passing Interface Standard,* Message Passing Interface Forum, http://www.mpi-forum.org/docs/mpi-11.ps, June 1995.

[19] S. Pakin, "Reproducible Network Benchmarks with CONCEPTUAL," *Proc. 10th Int'l Euro-Par Conf.,* pp. 64-71, 2004, http://www.c3.lanl.gov/pal/publications/papers/Pakin2004:reproducible.pdf.

[20] S. Pakin, "Rapid Development of Application-Specific Network Performance Tests," *Proc. Int'l Conf. Computational Science (ICCS '05) Workshop Tools for Program Development and Analysis in Computational Science,* http://www.c3.lanl.gov/pal/publications/papers/Pakin2005:conc-library.pdf, May 2005.

[21] S. Pakin, "CONCEPTUAL User's Guide," Technical Report LA-UR 03-7356, Los Alamos Nat'l Laboratory, Los Alamos, New Mexico, http://www.c3.lanl.gov/pakin/software/conceptual/conceptual.pdf, Oct. 2003.

[22] S.J. Deitz, B.L. Chamberlain, and L. Snyder, "Abstractions for Dynamic Data Distribution," *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS '04), Ninth Int'l Workshop High-Level Parallel Programming Models and Supportive Environments (HIPS '04),* pp. 42-51, http://www.cs.washington.edu/research/zpl/papers/data/Deitz04.pdf, Apr. 2004.

[23] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Unif Orm Pseudorandom Number Generator," *ACM Trans. Modeling and Computer Simulations,* vol. 8, no. 1, pp. 3-30, http://www.math.keio.ac.jp/ nisimura/random/doc/mt.ps, Jan. 1998.

[24] A. Hoisie, O.M. Lubeck, and H.J. Wasserman, "Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters," *Proc. Seventh Symp. Frontiers of Massively Parallel Computing (Frontiers '99),* pp. 4-15, http://www.c3.lanl. gov/pal/publications/papers/Hoisie1999:Sweep3D.pdf, Feb. 1999.

[25] W.R. Stevens, B. Fenner, and A.M. Rudoff, "The Sockets Networking API," *Unix Network Programming,* vol. 1, third ed., Addison-Wesley, Nov. 2003.

[26] E.R. Gansner and S.C. North, "An Open Graph Visualization System and Its Applications to Software Engineering," *Software—Practice and Experience,* vol. 30, no. 11, pp. 1203-1233, http://www.research.att.com/sw/tools/graphviz/GN99.pdf, Sept. 2000.

[27] H. Voß, *PSTricks: Grafik mit PostScript für* TEX *und* LATEX. Lehmanns Fachbuchhandlung, 2005.

[28] E. Strohmaier and H. Shan, "Apex-Map: A Global Data Access Benchmark to Analyze HPC Systems and Parallel Programming Paradigms," *Proc. ACM/IEEE Supercomputing Conf. (SC '05),* http://sc05.supercomputing.org/schedule/pdf/pap280.pdf, Nov. 2005.

[29] Information Networks Division, *Netperf: A Network Performance Benchmark, Revision 2.1,* Hewlett-Packard Company, http://www.netperf.org/netperf/training/netperf.ps, Feb. 1996.

[30] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang,* second ed., Prentice Hall, http://www.erlang.org/download/erlang-book-part1.pdf (part 1 only), Jan. 1996.

[31] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kini, D.K. Panda, and P. Wyckoff, "Microbenchmark Performance Comparison of High-Speed Cluster Interconnects," *IEEE Micro,* pp. 2-12, http://nowlab.cis.ohio-state.edu/publications/journal-papers/2004/liuj-ieeemicro04.pdf, Jan.-Feb. 2004.

[32] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology," *IEEE Micro,* vol. 22, no. 1, pp. 46-57, http://www.c3.lanl.gov/pal/publications/papers/petrini02:qsnet-micro.pdf, Jan.-Feb. 2002.

Scott Pakin received the BS degree in mathematics/computer science with research honors from Carnegie Mellon University in May 1992, the MS degree in computer science from the University of Illinois, Urbana-Champaign, in January 1995, and the PhD degree from the University of Illinois, Urbana-Champaign, in October 2001. Since 2002, he has worked as a technical staff member in the Performance and Architecture Lab (PAL) at the Los Alamos National Laboratory. His current research interests include analyzing and improving the performance of high-performance computing systems with particular emphasis on the communication subsystem. He has published papers on such topics as high-speed messaging layers, language design and implementation, job-scheduling algorithms, and resource-management systems.